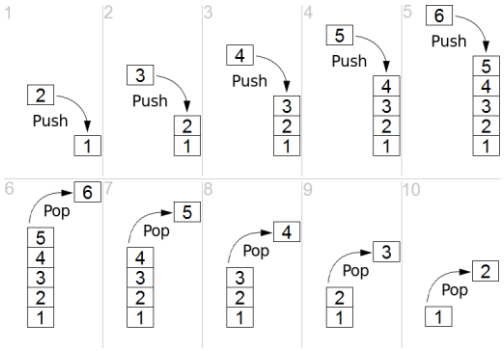
ITCS 209 Object Oriented Programming	Name:	Lab Score	Challenge Bonus
	ID:		
	Section:		

Lab07: Inheritance

Introduction:

One of the powerful mechanisms of OOP is inheriting features (methods and class variables) from existing classes, saving time and effort for programmers. In this lab, you will get hands-on experience using such a mechanism by implementing a generic class for a stack data structure that extends `ArrayList`.

Stack:



A stack is an abstract data type that serves as a collection of elements, with three principal operations:

1. `push(obj)`, which adds an element, `obj`, to the collection.
2. `pop()`, which removes the most recently added element that was not yet removed.
3. `peek()`, which gives access to the top element without modifying the stack.

The push and pop operations can be visualized in the above figure.

ArrayList:

`ArrayList` is a Java built-in implementation of the list data structure. In the previous lab, you have learned how to use an `ArrayList` object to store a set of objects of arbitrary types into a list. `ArrayList` provides useful interfaces such as `add(obj)`, `get(index)`, and `size()`.

Ok, here's what you have to do:

In this time-limited lab assignment, you are asked to implement class `Stack` of a generic type `<T>` in `Stack.java`. That is, your stack should be able to store objects of an arbitrary type. You have two options to implement `Stack<T>`:

> **Option 1**: Implement a generic class `Stack` of type `<T>` by extending `ArrayList`.
>
> **Option 2**: Implement a generic class `Stack` of type `<T>` from scratch, without extending any classes.

So choose wisely—you only have two hours in the lab, so Option 2 obviously does not seem to be a feasible solution! Hint: Option 1's solution contains fewer than 60 lines of readable code!

Your class should support two types of stacks: capacity-limited and -unlimited stacks. At the minimum, your class `Stack<T>` should support the following operations:

`public Stack()` is the default constructor to create an empty stack, with **no capacity limitation**.

`public Stack(int capacity)` is a constructor that takes an `int` input "capacity" which determines the **maximum number of elements that this `Stack` can contain**.

`public boolean push(T object)` pushes "`object`" of type `T` into the stack. Returns `true` if the object is successfully pushed, `false` otherwise (i.e., stack is full).

`public T pop()` pops and returns the top-most element in the stack. Returns `null` if the stack is empty.

`public T peek()` returns the top-most element in the stack without removing it. Returns `null` if the stack is empty.

`public T[] toArray()` returns the array representation of the stack, where the top element becomes the first element in the array. Return null if the stack is empty.

`public String toString()` overrides the existing `toString()` method. Use your creativity to display your stack in an intuitive manner. See the example in the sample outputs.

Testing your code:

StackTester.java is provided for you to test your code. Expected outputs are given on the next page.

Challenge Bonus (Optional): The Tower of Hanoi

Create ToH3 class (in ToH3.java) with the main method doing the following:

1. Create three stacks of Integer: s1, s2, s3.
2. Initialize s1 by *pushing* 3, 2, and 1, respectively, so that the elements are in ascending order:

s1	s2		s3
1 2 3			

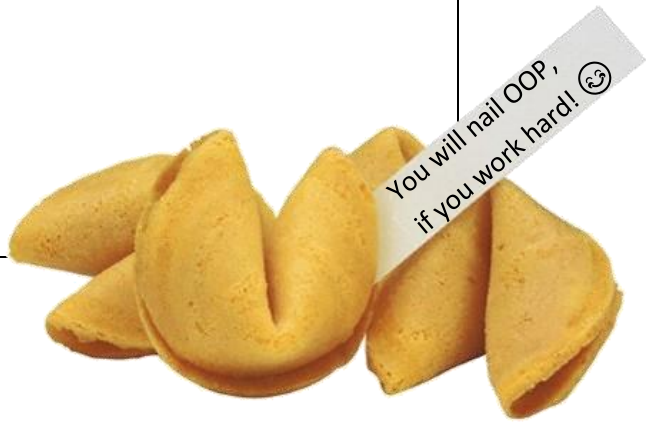
3. Display step-by-step, how you can utilize only the operations `push` and `pop` of s1, s2, and s3 to move 1, 2, and 3 from s1 to s3, ***with the condition that on a given stack, a smaller number must be on top of a bigger number***. The final stage of the three stacks should appear:

s1	s2	s3
		1 2 3

How many steps do you take?

Sample outputs:

***testCharacter*(); Output**	***testString*(); Output**
Pushing A |A| === Pushing B |B| |A| === Pushing C |C| |B| |A| === Pushing D |D| |C| |B| |A| === Pushing E |E| |D| |C| |B| |A| === Stack Full:[A, B, C, D, E] The top element is E. Cannot push F |E| |D| |C| |B| |A| === Popping E |D| |C| |B| |A| === Popping D |C| |B| |A| === Popping C |B| |A| === Popping B |A| === Popping A === Stack empty! ===	Pushing Think? |Think?| === Pushing You |You| |Think?| === Pushing Don't |Don't| |You| |Think?| === Pushing Cool |Cool| |Don't| |You| |Think?| === Pushing is |is| |Cool| |Don't| |You| |Think?| === Pushing Java |Java| |is| |Cool| |Don't| |You| |Think?| === Pushing Oops! |Oops!| |Java| |is| |Cool| |Don't| |You| |Think?| === Popping Oops! |Java| |is| |Cool| |Don't| |You| |Think?| === Popping Java |is| |Cool| |Don't| |You| |Think?| === Popping is |Cool| |Don't| |You| |Think?| === Popping Cool |Don't| |You| |Think?| === Popping Don't |You| |Think?| === Popping You |Think?| === Popping Think? === Stack empty! ===



Appendix: Implementing a Generic Class

A generic class <T> allows your class to work with any data types. ArrayList is a generic class since you can create an ArrayList of any types of objects. i.e.

```
ArrayList strList = new ArrayList<String>();      // An array list of String objects
ArrayList intList = new ArrayList<Integer>();     // An array list of Integer objects
```

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) `T1`, `T2`, ..., and `Tn`.

For example, you can create a *generic type declaration* for a Box class by declaring "`public class Box<T>`". This introduces the type variable, `T`, that can be used anywhere inside the class.

With this change, the `Box` class becomes:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of `Object Type` are replaced by `T`. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

Learn more here: https://docs.oracle.com/javase/tutorial/java/generics/types.html